Developing IEEE1394-FireWire driver for OS/2-eCS

Alexandr Cherkaev
doctor64@gmail.com

Topics

- quick introduction to firewire. History, bus architecture overview, advantages.
- Firewire stack for os/2: functions, calling conventions, how it works together.
- Build environment.
- Writing driver for firewire stack.
- current state of project and plans

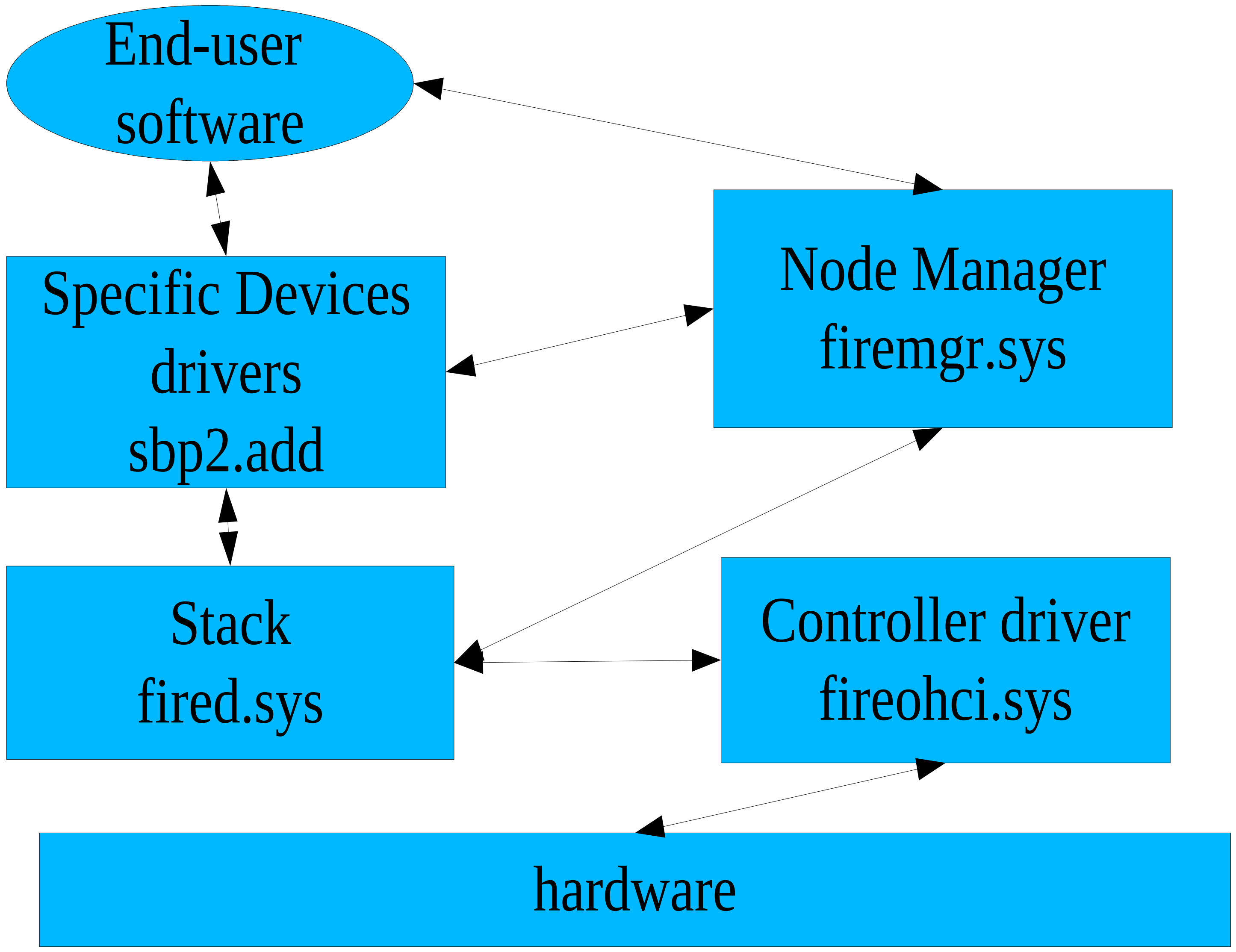
History of FireWire

- developed in mid-90
- 1394-1995 standard
  - first generally available
- 1934a standard
  - extended power management, fixed some performance and interoperability issues
- 1934b standard
  - faster speeds up to 3,2Gb/s

Facts about FireWire

- up to 63 device on bus
- up to 1023 buses
- speed up to 400 Mb/s
- hot plug
- all devices are equal
- asynchronous transfer
- isochronous transfer
- 1394b add speeds
  - 800 Mb/s, 1,6 Gb/s, 3,2Gb/s
- 64 bit addressing
- CSR architecture
- peer-to-peer transfers

Firewire stack design



FireWire for OS/2 presentation
In this presentation I present Firewire for os/2-eCS project. The following topics will be covered:
1: quick introduction to firewire. History, bus architecture overview, advantages.
2: Firewire stack for os/2: functions, calling conventions, how it works together.
3: Build environment.
4: Writing driver for firewire stack.
5: current state of project and plans

1)
Firewire bus developed by Apple, Motorola and IBM in 90s. Bus was developed as faster and more advanced alternative to USB. Some characteristics of firewire:
up to 63 device on each bus, up to 1024 buses, speed up to 400 mb/s, hot plug and plug-and-play, support for asynchronous and isochronous transactions.
Bus history:
1394-1995 standard. first generally available
1394a extends 1995 standard in term of power management and some interoperability and performance issues
1394b add 800 Mb/s, 1.6 Gb/s and 3.2 Gb/s speeds

Hardware
We don't need to have deep knowledge in hardware details, just few words about firewire specific hardware
devices have one or more ports, so no need in special hub device, unlike USB
all devices are equal, no definitely specified master and slaves. All devices are equal in term of bus management, root of bus tree is selected during bus reset process. When any device attached/detached to bus, initiated bus reset.
Device can do peer-to-peer data transfers, without intervention of host system
Some notes on terminology:
Module – physical device attached to bus, contain one or more nodes
Node – represent logical entry within module. Nodes have own address on bus and contain Control and Status Registers (CSR) and ROM entires. Some node registers are shared between units within node.
Unit – functional subcomponent of a node. Units within node typically operate independently and controlled by their own software drivers.
Asynchronous transaction- provide error-free mechanism to deliver data and got acknowledgment, but not guarantee time in which packet will be delivered.
Isochronous transactions guarantee bandwidth.

During bus reset process nodes receive their address, called physical ID. Combination of bus address and Physical ID form NodeID, which universally identify each node within system. Also selected Root Node, filled and broadcasted Speed map and Topology map, which describe bus topology and speed characteristics of nodes on bus. Selected managers, who control bus work – cycle master, isochronous resource manager, bus

manager.
Cycle master provide clock for isochronous transactions
Isochronous resource manager control isochronous resource allocation
Bus manager fill and publish for other nodes speed and topology map of bus, and provide power management on bus.

Now, about os/2 realization.

Its consist next main parts:
driver for firewire card. It cover all physical card activity. Currently driver for OHCI based cards is written and, with some limitations worked. Of course, lots of thing need to be done – mostly support for power management and isochronous transactions.
next part – and heart of all firewire stack – fired.sys. It cover packet queuing, delivering to client drivers, processing bus reset and other bus events. All other drivers connect to it.
Next – node manager. firemgr.sys is cover tasks of building information structures which describe all devices available on bus and give API to device specific drivers.
So, we have following model:
(slide 1)
Next, about inter driver communications.
Current realization is not a good design, but i must use this approach to be compatible with Linux realization.

First of all, fired.sys stack driver is loaded and initialized. Next, other drivers via IDC get address of big, ugly structure, contained pointers to externally callable functions. Some thing occur when drivers register in stack. Later, drivers can do direct FAR CALL to code located in other drivers.
when Nodemgr loaded, it register as highlevel driver with *fpfnRegisterHighlevel() call (hpsb_register_highlevel in Linux)
This call passes a structure containing pointers to four functions
(*fpfnAddHost, *fpfnRemoveHost, *fpfnHostReset and *fpfnIsoReceive) provided by the registering high-level driver (protocol). These functions are then called when
the appropriate event occurs in the underlying system, eg. *fpfnAddHost is
called every time a new host (a 1394 interface card) is detected and
initialized, and *fpfnIsoReceive is called every time isochronous data is
received from the low-level drivers.
When *fpfnRegisterProtocol is called, *fpfnAddHost is
immediately called for every host already known by the ieee1394 stack.
High-level protocol drivers can also register themselves as responsible for
handling read/write/lock transactions for areas of the 1394 address space on
the local hosts. To do so, they must call the function
*fpfnRegisterAddrspace (hpsb_register_addrspace in Linux). This call passes
a structure containing the address range and four pointers (*read, *write,
*lock and *lock64) provided by the registering high-level driver. These
functions are then called when the appropriate event occurs in the underlying

system, eg. *read is called when the system receives a read request for an address in the specified range.
When calling the two mentioned registration functions, it is allowed to let some of the pointers be null pointers. If for example only *read was non-null in a *fpfnRegisterAddrspace call, the address range in question will be ``read only" from the 1394 bus. Likewise, letting eg. the *iso_receive pointer in a hpsb_register_highlevel call be null will prevent the high-level driver from being called upon reception of isochronous data.
The 1394 standard uses a 64 bit address space, where the upper 16 bits represents the node-ID. High-level drivers register for the lower 48 bits of the address space. This means that a high-level driver will be called for request for the specified addresses on all the local hosts. The High-level driver can differentiate the handling of the hosts if required, as a reference to the host in question is passed along with the call.
Next, drivers can attach to node manager and will be notified when devices, conforms to specific requirements is attached to bus. It can be device id, so we can have driver for very specific device, driver for protocol, so we can have driver for class of device, f.e. sbp2 driver can handle all devices which conform to SBP2 standard. we call fpfnRegisterProtocol() with parameters like this:
match_flags= IEEE1394_MATCH_SPECIFIER_ID| IEEE1394_MATCH_VERSION;
specifier_id=SBP2_INIT_SPEC_ID_ENTRY|0xffffff;
version_id=SBP2_SW_VERSION_ENTRY|0xffffff;
later, node manager will call back the specified in structure function *probe() when device appear on bus, *update() each time when device survive Bus Reset, *remove() when device removed.
Finally, all highlevel drivers use stack services to create packets and communicate (read/write) devices on bus.
of course, remembering what we using 16 bit model, all addresses passed between drivers must be FAR pointers.
Build environment:
I use Watcom for building firewire stuff. of course, DDK is needed. Also used wpddlib from Netlabs.
here is sample of simple driver skeleton which use firewire stack function initialization:
1: do standard os/2 device driver initialization, like saving DevHelp pointer for later use Initialize driver internals as well.
2: use IDC to attach to fired.sys and get pointers to stack functions and data.
3: use IDC with fired.sys to get node manager structures and data.
4: register in stack as highlevel driver and register address space with stack driver.
5: register in node manager as driver for specific device and/or device class.
Work:
when new host (bus controller) is detected, driver got fpfnAddHost() from stack and can do needed internal work, like creating data structures for new host.
When host removed – driver got another callback – fpfnRemoveHost()

When device plugged/unplugged or software initiated bus reset occur, driver got fpfnBusReset() callback. Driver must take care about processed device address on bus can be changed.
Driver receive callbacks when device communicate with driver by issing read/write/lock transactions to driver address space registers.

Driver also can get callbacks from NodeManager, when specified in *fpfnRegisterProtocol() call device is appear/remove/change on bus.

And, of course, driver can communicate with device by device specific way, issuing appropriate read/write requests to device when applications requests.

Present and future
Currently, all stack drivers working ok with asynchronous transaction. Nodemanager is working, developing of sbp2 driver (mass storage driver) is in progress.
Also exist utility from Robert Henschel, to access nodemanager and browse attached devices info.
plans for project:
os/2 resource manager support
isochronous transactions support (needed for mini-dv camcoders)
port more drivers. For video cameras, ethernet over 1394, audio, scanners and other.
PCMCIA/CardBus support in fireohci driver.
port libraw to give ring3 applications access to device on firewire
change internal interfaces from direct calls to IDC.